Journal of Institute of Control, Robotics and Systems (2011) 17(1):38-46
DOI:10.5302/J.ICROS.2011.17.1.38 ISSN:1976-5622 eISSN:2233-4335

Windows NT 상에서의 OPRoS 컴포넌트 스케줄러의 실시간성 분석 및 개선

Real-Time Characteristics Analysis and Improvement for OPRoS Component Scheduler on Windows NT Operating System

이 동 수, 안 희 준*
(Dong-Su Lee[1] and Heejune Ahn[1])
[1]Seoul National University of Science and Technology

Abstract: The OPRoS (Open Platform for Robotic Service) framework provides uniform operating environment for service robots. As an OPRoS-based service robot has to support real-time as well as non-real-time applications, application of Windows NT kernel based operating system can be restrictive. On the other hand, various benefits such as rich library and device support and abundant developer pool can be enjoyed when service robots are built on Windows NT. The paper presents a user-mode component scheduler of OPRoS, which can provide near real-time scheduling service on Windows NT based on the restricted real-time features of Windows NT kernel. The component scheduler thread with the highest real-time priority in Windows NT system acquires CPU control. And then the component scheduler suspends and resumes each periodic component executors based on its priority and precedence dependency so that the component executors are scheduled in the preemptive manner. We show experiment analysis on the performance limitations of the proposed scheduling technique. The analysis and experimental results show that the proposed scheduler guarantees highly reliable timing down to the resolution of 10ms.

Keywords: OPRoS (Open Platform for Robotic Service), windows NT, real-time system, task scheduling

I. 서론

최근 서비스로봇에 대한 개발과 상용화에 대한 관심이 증가하고 있다[1]. 대표적인 서비스 로봇 개발의 예로 국외에서는 카네기멜론 대학의 Minerva [2], Honda사의 Asimo [3] 등을 들 수 있고, 국내에는 KIST의 마루, 유진로봇의 IRobi-Q, 한울의 TRIO, 다사로봇의 FRIDA등 다양한 로봇들이 개발되고 있다.

서비스 로봇의 중요한 두 가지 특징은, 기존의 산업용 로봇에 비하여 다양한 '서비스'를 제공하여야 한다는 점과, 대부분의 서비스 로봇이 인간과 상호작용을 하는 특징으로 인하여 실시간성이 요구된다는 점을 꼽을 수 있다. 위에서 예로든 로봇들의 사양과 기능들을 분석해 보면, 이동과 작동에 관련된 제어기능과, 환경을 인식하는 센서에 관련된 요소들, 그리고 사용자와 인터페이스를 하기 위한 음성, 영상신호처리를 중심으로 한 HRI 기능을 주요 요소로 하고 있으며, 이를 바탕으로 다양한 서비스 제작하여 구성할 수 있도록 하고 있다.

서비스 로봇의 구현기술의 핵심은 사용자에게 도움이 되는 새로운 서비스를 지속적으로 제공하는 것이다. 따라서 하나의 개발업체에서 모든 응용을 제공하기에는 한계가 있다. 따라서, 로봇 응용 제작을 하는데 있어서 모듈화, 재사용성과 생산성의 장점을 가진 컴포넌트 기반의 프레임워크들이 개발되고 있다. 대표적인 예로 국내의 OPRoS (Open Platform for Robotic Service) [4], OMG의 RTC (Robot Technology Component) [5], 그리고 Microsoft의 MSRDS (Microsoft Robotics Developer Studio) [6]가 컴포넌트 구조에 기반을 하고 있다.

이러한 프레임워크와 함께, 많은 개발자 확보를 위한 편리한 개발환경이 제공되어야 함은 물론이고, 다양한 응용을 손쉽게 개발할 수 있는 환경이 필요하다. 이러한 이유 때문에 현재 상당수 서비스 로봇은 범용 운영체제인 Windows NT 기반이나 리눅스 기반의 운영체제를 사용하고 있으며, 실시간 운영체제(RTOS)를 채용하는 경우는 아주 드물다. 그 근본원인은 범용운영체제를 사용함으로 인해 많은 개발자를 비교적 쉽게 확보할 수 있고, 운영체제가 제공하는 많은 라이브러리 및 유틸리티를 통하여 다양한 응용을 쉽게 만들 수 있기 때문이다.

반면, 서비스 로봇의 기계 및 제어적인 요소를 포함하고 사람이 생활하는 공간에 동작하기 때문에, 로봇의 운영환경은 실시간성 지원이 필수적이다. Windows나 리눅스 기반의 범용 운영체제는 전체적인 처리의 효율성을 목적으로 하고 있으며, 따라서 라운드로빈 기반의 스케줄링, 동적 우선순위의 스레드, 상대적으로 낮은 해상도의 타이머 등의 특성을 갖고 있다. 따라서 일반적으로 실시간성 지원이 쉽지 않다.

이처럼 범용 운영체제를 서비스로봇에 사용하였을 때의 장단점을 이용하고 극복하기 위하여, 본 연구는 별도의 수정 없이 Windows NT 운영체제가 지원하고 있는 실시간성 특징[7]을 이용하여 현재 국내 서비스 로봇의 표준 프레임워크로 개발중인 OPRoS 프레임워크를 운영하기 위한 방안에 대하여 제안한다.

* 책임저자(Corresponding Author)
논문접수: 2010. 8. 23., 수정: 2010. 12. 22., 채택확정: 2010. 12. 26.
이동수, 안희준: 서울과학기술대학교 제어계측공학과
(acemania83@hanmail.net/Heejune@seoultech.ac.kr)
※ 본 논문은 2010년도 ICROS 학술대회에서 초안이 발표되었음.

이러한 범용 운영체제 중 Windows NT는 1993년에 발표된 이후로 Windows 2000, Windows XP, Windows Vista 및 최근 발표된 Windows 7의 기본 커널로 사용되며, PC용 운영체제의 과반수를 차지하고 있다. 앞서 말한 범용운영체제 가지고 있는 장점과 서비스 로봇의 실시간성 지원이라는 요구사항을 모두 만족하기 위해 Windows 환경에서의 기존의 연구들이 있어왔다[7-11]. 한가지 방법은 RTX [12]를 지원하는 NT의 디바이스 드라이버를 확장하여 커널의 기능을 확장하는 방법이다. 이 방법은 매우 높은 실시간성을 필요로 하는 경우 물론 효율적이라 생각하지만, 앞서서 말한 범용 운영체제가 가지는 장점을 제한하는 결과를 가져오게 된다. 본 연구는 별도의 커널 수정 없이 Windows NT 운영체제가 지원하고 있는 실시간성 특징[7]을 이용하여 현재 국내 서비스 로봇의 표준 프레임워크로 개발중인 OPRoS 프레임워크를 운영하기 위한 방안에 대하여 제안한다. 본 연구에서는 주기적 실행주기를 갖는 OPRoS 컴포넌트만을 대상으로 하는 사용자 레벨의 스케줄러를 설계하고, 실험을 통하여 그 효율성을 확인하고, 지원한계를 알아보려고 한다.

서론에 이어 제 II 장에서는 우선 OPRoS 컴포넌트 모델에 대하여 약술한다. 특히 스케줄링과 관련된 특징을 소개하고 문제점에 대하여 설명한다. 제 III 장에서는 Windows NT 운영체제가 가지는 실시간성 지원 특성과 저해 요인에 대하여 분석하고, Windows NT가 가지는 한계점에 대하여 설명한다. 제 IV 장에서는 이러한 특징과 한계점을 고려하여 스케줄러의 구조를 제안한다. 제 V 장에서는 제안된 스케줄러를 기반한 다양한 실험을 통하여 그 효율성을 검증한다. 제 VI 장에서는 본 연구의 의미와 결론에 대해 설명한다. 마지막으로 제 VII 장에서는 제안된 스케줄러를 코드로 제시하면서 마무리 한다.

II. OPRoS 컴포넌트 모델의 소개

1. OPRoS 컴포넌트 모델

OPRoS 프레임워크의 핵심이 되는 컴포넌트 모델은 2009년 한국 로봇협회에서 표준 규격[4]이 되었다. 또한 참고구현[13], 그리고 OPRoS의 초안의 설명 등이 공개되었다. 이에 대하여 간단히 설명하면 다음과 같다. 그림 1은 OPRoS 시스템의 전체적인 구조를 간략히 도식화 하고 있다. OPRoS 시스템은 보통 컨테이너라고 불리는 런타임 프레임워크와 컴포넌트들로 구성되어 있다. 컴포넌트는 각각 특정적인 센서, 구동기, 알고리즘 등의 응용서비스를 제공하며, 이 컴포넌트들간의 정보교환은 포트를 통해서만 가능하다. OPRoS 규격은 호출의 동기화 또는 비동기화 여부 및 전달데이터의 특성에 따라 데이터, 이벤트, 서비스 포트의 3가지 포트를 제공하고 있다.

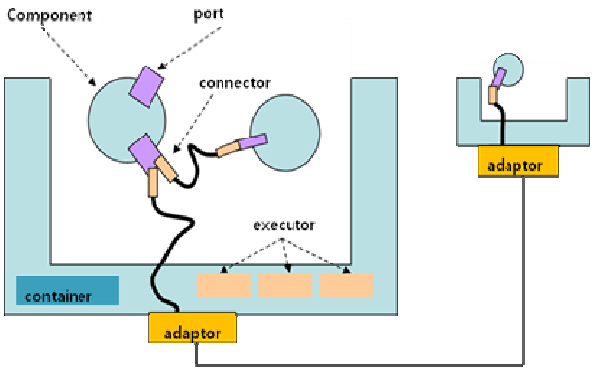


그림 1. OPRoS 프레임워크의 구조와 컴포넌트 모델.
Fig. 1. Structure of OPRoS framework and component model.

각 컴포넌트는 'Component'라고 하는 부모 클래스를 상속받아 구현하도록 정의하고 있으며, 이를 사용자 정의 컴포넌트라고 부른다. 컴포넌트 개발자는 이 사용자 정의 컴포넌트와 다수의 보조 클래스를 통하여 서비스를 제공하며, 해당 컴포넌트의 구성 정보는 '컴포넌트 프로파일'이라고 부르는 XML 파일에 정의하도록 하고 있다. 보통은 몇 개의 컴포넌트들이 합쳐져서 하나의 태스크, 또는 응용서비스를 구성하게 된다. 이러한 구성내용과 설정내용들은 '응용 프로파일'이라고 부르는 XML 파일을 통하여 정의되도록 하고 있다. 프레임워크와 컴포넌트간의 인터페이스는 기저 Component 클래스의 인터페이스, lifecycle, port, property 인터페이스를 통하여 이루어 진다. 프레임워크는 콜백함수인 initialize(), start(), stop(), destroy(), recover(), update(), reset() 등을 호출하여 컴포넌트의 생성과 소멸, 구성, 스케줄링, 통신서비스 등을 지원하게 된다.

2. OPRoS 스케줄링 방식의 특징과 문제점

OPRoS의 스케줄링 방식은 크게 주기적(periodic)인 형태와 비주기적(aperiodic)인 형태로 구분된다. 본 연구에서는 주기적인 형태의 스케줄링만을 고려한다. 제어 시스템에서 주기적인 수행구조는 매우 일반적인 처리 구조이다. 이는 제어 시스템이 일반적으로 센싱, 알고리즘, 액츄에이션의 회귀적인 구조를 갖는데, 디지털 제어의 경우는 이에 대하여 일정한 샘플링 주기로 반복적인 수행을 하게 되기 때문이다[14]. 현재 공개된 OPRoS 참고구현의 스케줄링 알고리즘 특징을 간단히 살펴보면 다음과 같다.

첫째, 각각의 실행기들은 주기에 따라 각각 동적 우선순위인 일반 스레드로 생성되어 동일한 주기의 컴포넌트들을 실행시킨다. 이때 실행기들은 운영체제에 의하여 스케줄링 받으며 실행기들 사이의 우선순위는 존재하지 않는다.

둘째, 실행기들은 한 주기 동안의 실행이 완료되면 실행시간을 측정하여 자신의 주기에서 실행시간을 뺀 나머지 시간을 Sleep하게 된다.

위와 같은 알고리즘의 문제점은 다음과 같다. 첫째, 동적 우선순위의 일반 스레드로 생성되기 때문에 커널에 의해 우선순위가 변경될 가능성이 있다. 둘째, 스레드간 우선순위가 존재하지 않기 때문에 긴 주기와 긴 실행시간을 갖는 실행기에 의해 짧은 주기를 갖는 실행기의 주기성이 깨질 가능성이 존재한다. 셋째, 실행이 완료된 후 실행시간을 측정하기 때문에, 시간 측정함수와 Sleep 함수의 해상도에 따라서 주기성이 보장되지 않을 수 있다.

III. Windows NT의 실시간 특성

1. Windows NT의 실시간 지원 특성

실시간성을 지원하기 위해 OS가 지원해야 하는 기능은 첫째, 선점형 멀티 스레드 스케줄링 지원, 둘째 확정적 스레드 우선순위 지원, 셋째, 확정적인 스레드 동기화 방식, 넷째, 우

선순위 상속 지원이다[7].

Windows NT는 실시간 운영체제가 아님에도 불구하고 그 특성을 살펴보면 위의 요건 중 우선순위 상속을 제외하고는 실시간 운영체제로서의 특성을 상당부분을 만족하고 있다[7]. 우선, 비선점형과 선점형 사용자 스레드를 모두 제공하고 있으며, 32개의 우선순위를 제공한다. 이중 특히 16~31까지의 우선순위는 실시간성에 적합한 정적 우선순위(static priority)를 지원한다. 또한 임계영역(critical section), 이벤트(event), 세마포어(semaphore), 뮤텍스(mutex) 및 스레드 제어 함수들을 통하여 확정적인 스레드 동기화 제어가 가능하다. 또한 최소 1ms까지의 해상도를 지원하는 멀티미디어 타이머(multimedia timer)를 제공하고 있어 최소 1ms의 타이머 인터럽트가 제공가능하다[8].

2. Windows NT의 실시간 지원의 한계

Windows NT는 실시간성을 지원하는 특성을 다수 가지고 있는 반면, 실시간 운영체제로서 적합하지 못한 특성을 가지고 있다. 우선 특정 스레드가 주기적으로 스케줄링 되는데 발생할 수 있는 문제 요인을 그림 2에 도식화 하였다.

첫째로 타이머의 정밀도에 의하여 실제 예정된 시간보다 지연되어 스케줄링 될 수 있다. Windows NT는 HAL (Hardware Ab-straction Layer)를 이용하여 OS 타이머 인터럽트(OS timer interrupt)를 발생시킨다. [8]에 따르면 일반적인 상황에서 OS 타임 틱(OS time tick)은 약 10~15ms에 발생되고, 이 시간은 'timeBeginPeriod()'함수를 통하여 1ms까지 줄일 수가 있다. 하지만 HAL에서 사용되는 RTC (Real Time Clock)의 특성상 1ms로 설정하더라도 실제 인터럽트가 발생하는 시간은 0.976ms가 되고, 또한 약 1ms 정도의 오차도 존재 하게 된다[8]. 둘째로 ISR (Interrupt Service Routine)과 DPC (Deferred Procedure Call), 그리고 커널 스레드에 의한 지연이다. Windows NT는 인터럽트 실행시간을 최소화 하는 방안으로 대부분의 처리를 DPC로 넘기고, 인터럽트처리와 사용자 스레드 수행 사이에 처리하도록 하고 있다. 즉, 현실적으로 우선순위가 'ISR > DPC > 커널 스레드와 실시간 스레드 > 비 실시간 스레드' 가 되는 것이다[15]. 이때, 일반적으로 ISR 시간은 매우 작으므로 무시한다면, DPC에 의한 지연이 스케줄링의 주기성을 방해하는 주요 요소가 된다. DPC에 의한 지연은 USB 데이터전송, 네트워크 트래픽, Disk IO 데이터 등의 처리이다. 셋째로, Windows NT에서 실행되고 있는 여러 프로세스들과 스레드 등의 실행에 의한 지연이 존재 가능하다. 우리는 5절의 실험결과를 토대로 처음 두 가지 요소의 지연 정도와 한계에 대하여 설명한다. 하지만, 마지막 요소의 경우는 NT가 선점성을 지원하기 때문에 다른 실시간 운영체제와 차이가 없는 부분으로 수행 수락 알고리즘과 스레드간의 우선순위 등으로 해결해야 하는 부분이다. 따라서, 처음 두 가지 요소의 지연 오차범위가 허용가능 하다면 실시간성 보장이 어느 정도 가능하다고 판단된다.

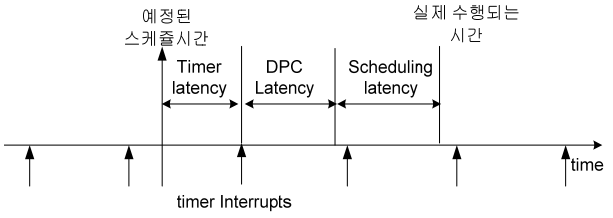


그림 2. 스케줄링의 주기성 지연 요인

Fig. 2. Delay elements of Windows NT task scheduling.

또한, 우선순위 상속기능이 없는 관계로 우선순위 역전현상을 해결하지 못하는 문제가 존재하며, 마지막으로 문제시되는 사항은 실시간 우선 순위가 실제로는 16, 22, 23, 24, 25, 26, 31. 즉 7개로 상당히 제한적이라는 점이다[7].

IV. OPRoS 스케줄러 설계

1. 현 OPRoS 참고구현의 스케줄링 기법

그림 3은 현재 OPRoS 참고구현의 스케줄링 기법을 도식으로 표시한 것이다. 각 실행기는 동일주기를 갖는 컴포넌트를 모아서 콜백함수를 호출함으로써 실행을 하며, 첫 컴포넌트의 호출 직전 시각 (si) 과 마지막 컴포넌트의 호출직후 시간 (ei)를 측정하고, 이 실행간격을 실행주기 T에서 제외하고 Sleep 함수를 호출한다. 즉 $T_{sleep} = T_{period} - (e_i - s_i)$ 가 되는 것이다.

현재 구현이 갖고 있는 문제점은 다음과 같이 정리할 수 있다. 실행기 스레드에 동적-비선점 우선순위 스레드를 사용하기 때문에 확정적인 우선순위 지원이 불가능 하고, 각 실행기들 사이에 우선순위가 존재하지 않기 때문에 어떤 실행기가 먼저 수행될지 알 수 없다. Sleep()함수, 시간 측정함수의 해상도 한계에 의해 정밀한 타이밍 지원이 불가하다. 또한 각 실행기들은 완전히 독립적으로 실행되므로 우선순위 고려 등이 불가능하며, 다중코어를 사용하는 시스템인 경우 컴포넌트간에 실행 상관성을 고려하지 못한다.

예를 들어 그림 3에서 실행기1은 실행기2보다 주기가 짧고 (T1 < T2), 따라서 일반적으로 높은 우선순위를 갖고 있어야 실시간성을 보장할 수 있다[17]. 그러나 현재는 실행기 사이에 우선순위가 보장되지 않기 때문에 3번째 주기(s3)에서 실행기2가 실행기1보다 먼저 실행되는 것을 막을 방법이 없으며, 이로 인해 실행기1의 컴포넌트들이 실행기한(deadline)안에 호출되지 못하는 경우가 생기게 된다.

2. 제안하는 컴포넌트 스케줄러

이러한 문제점을 해결하기 위하여 본 논문에서는 그림 4 와부록의 발췌 코드와 같이 사용자 레벨의 OPRoS 컴포넌트 스케줄러를 설계 하였다. 우선 NT의 확정적인 스케줄링을 위해, OPRoS 런타임프로세스는 Windows 운영체제상의 실시간 프로세스(REALTIME_PRIORITY_CLASS)로 설정하였고, 스케줄링 스레드는 실시간 우선순위 중 가장 높은 우선순위 (THREAD_PRIORITY_TIME_CRITICAL)의 스레드로 생성하

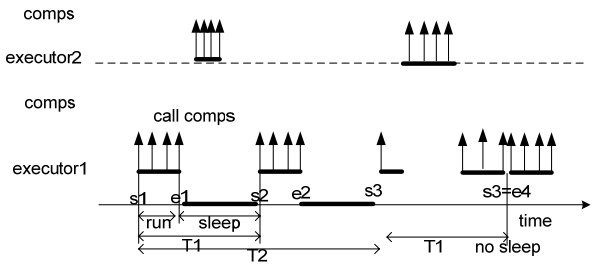


그림 3. OPRoS 참고구현의 실행기 스케줄링.

Fig. 3. Executors scheduling in OPRoS reference implementation.

였다. 또한 'timeBeginPeriod()' 함수를 사용하여, 현 시스템이 지원 가능한 최소의 시간해상도(1ms)를 지원하도록 설정하였고 멀티미디어 타이머를 통하여 10ms주기마다 인터럽트를 발생시켜 스케줄링 스레드가 깨어나도록 만들었다.

Windows NT가 제공하는 실시간 우선순위는 7개로 상당히 제한적이기 때문에, 실행기들의 우선순위를 Windows 운영체제의 우선순위에 매핑하는 것은 불가능하다. 따라서, 실행기 스레드는 운영체제상에서는 모두 동일한 우선순위를 갖도록 하고, 스케줄러 스레드가 이 실행기들 중 주기가 낮을수록 높은 우선순위로 판단하는 RMS (Rate Monotonic Scheduling)방식을 사용하였다[17,18].

스케줄러 스레드는 이벤트를 기다리고 있다가 깨어나서 스케줄링을 수행한다. 스케줄링 이벤트가 발생하는 시점은 멀티미디어 타이머에 의하여 주기적(여기서는 10ms)으로 깨어나는 경우와, 수행을 마친 실행기가 CPU를 양보할 목적으로 'yield_cpu()'함수 호출을 하는 경우이다. 스케줄러 스레드는 우선 멀티미디어 타이머에 의해 주기적으로 깨어나면(t1에 해당) 현재 준비상태(ready state)에 있는 실행기를 찾는다. 그림 4의 경우 스케줄링 스레드는 두 종류의 실행기 중 주기가 낮은 10ms 실행기를 항상 먼저 실행하게 된다. 스케줄러 스레드가 실행기를 실행시키는 경우에는 'ResumeThread()' API를 통하여 실행하게 된다. 그리고, 실행기가 모든 실행을 완료하면 실행기 스스로 'yield_cpu()' 함수를 호출하여 스케줄러 스레드를 깨운다. 'yield_cpu()'호출에 의하여 깨어난 스케줄러 스레드는 현재 동작 중이던 실행기를 'SuspendThread()' API를 통하여 정지시키고, 그 다음 낮은 주기의 실행기를 실행시킨다. 만약 그 다음 주기때(t2에 해당) 실행기가 수행 중에 있다 하더라도 스케줄러 스레드는 실행기들 보다 높은 우선순위를 가지고 있기 때문에 멀티미디어 타이머에 의해서 주기적으로 깨어나 CPU를 선점하게 된다. 이때 스케줄러는 현재 실행기 들의 상태를 검토하여 현재 실행중인 실행기보다 더 낮은 주기의 실행기가 준비상태에 있다면 직전 실행기는 suspend시키고, 더 낮은 주기의 실행기를 resume하게 된다(t2에 해당). 또한 현재 실행기가 다시 실행을 완료하여 CPU를 양보한 경우 스케줄러는 다시 그 다음 낮은 실행기를 실행시키게 된다(그림 5의 t2-1에 해당). 또한 준비상태의 실행기가 존재하지 않는 경우 다음 번 타이머 인터럽트가 발생 할 때까지 idle상태로 남아있는다. 추가적으로 OPRoS 표준에서는 컴포넌트간 실행의 순서를 유지해주도록 강제하고 있지만, 최근 다중코어 시스템이 범용화 되는 상황에서 동일한 응용이 다른 CPU에서 스케줄링 되는 경우 위와 같은 선행성을 유지하기 어렵다. 이 때문에 제안하는 스케줄러에서는 동일한 응용 프로파일에 해당하는 컴포넌트들은 'SetThreadAffinityMask()' API를 통하여 같은 CPU에 할당되도록 하였다. 또한, 실시간 컴포넌트들의 코드 영역과 주요데이터 영역은 page-lock 을 통하여 스와핑을 방지하였다.

제안하는 컴포넌트 스케줄러를 사용하면 2절에서 언급한 문제를 해결할 수 있다. 첫째, 각 실행기들은 실시간 우선순위 스레드로 생성되기 때문에 확정적인 스케줄링이 가능하다. 둘째, 스케줄러가 실행기의 주기를 가지고 우선순위를 판단하기 때문에 긴 주기와 긴 실행시간을 갖는 실행기에 의해

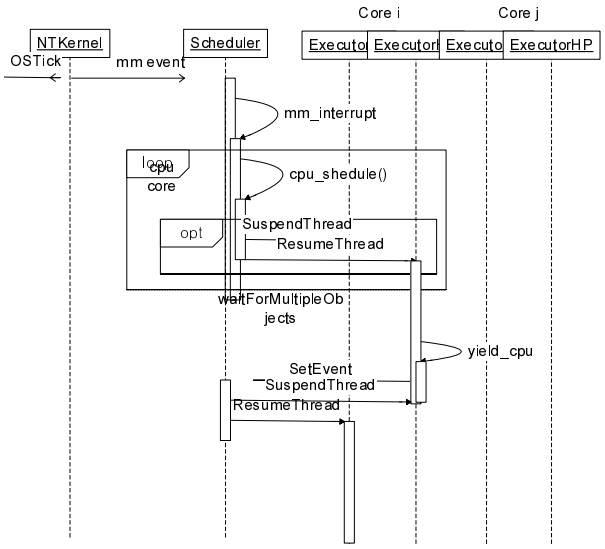


그림 4. 제안된 사용자 레벨 스케줄러의 함수 호출흐름.
Fig. 4. Call flow of the proposed component scheduler.

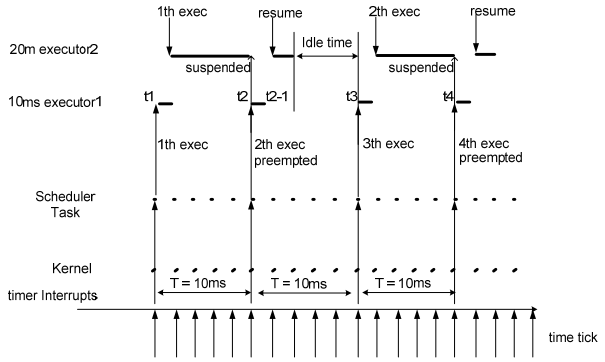


그림 5. 제안된 사용자 레벨 OPRoS 스케줄링 예.
Fig. 5. Proposed user level scheduler Example.

짧은 주기를 갖는 실행기의 주기성이 깨지는 경우를 방지 할 수 있다. 셋째, 높은 해상도의 멀티미디어 타이머 인터럽트를 사용하기 때문에 주기가 짧은 실행기들의 주기성도 보장될 수 있다.

V. 검증 실험

제안한 스케줄러의 성능을 확인하기 위하여 현재 공개된 OPRoS 참고구현의 스케줄링 방식과 제안하는 스케줄링 방식을 비교 실험 하였다. 앞서 3절에서 언급했던 스케줄링의 주기성 지연 요인 중 두 가지 요소인 타이머 인터럽트의 정확성에 대한 검증과 DPC 등의 외란(disturbance)이 타이머의 주기성에 주는 영향 정도를 판단하기 위한 실험을 수행하였으며, 또한 실제 제어 시스템에서 많이 사용되고 있는 DC 모터의 PI 위치 제어 실험을 통하여 실제 응용에서의 효용성도 검증하였다. 실험은 다양한 Windows 환경에서 수행하였다. 연구실에서 수집 가능한 노트북, 데스크탑, 서버 등에서 수행하였으며, 그 결과는 거의 동일하였다. 본 논문에서 제시한 데이터는 대표적으로 Intel Pentium Dual core CPU 2GHz, 2GHz Memory, Service Pack3에서 실험한 결과이다. 데이터 확보를 위한 시각측정은 Win32 환경에서 가장 정확한 시간 정보를

제공해 주는 Windows API 함수인 'QueryPerformanceCounter()', 'QueryPerformanceFrequency()'를 사용하였다. 데이터는 실행 중에 메모리에 저장하였고, 실행이 끝난 후 파일에 저장하여, 측정에 의한 영향을 최소화하였다.

1. 타이머 서비스의 정확성 검증 실험

그림 6은 10ms의 주기를 갖는 컴포넌트 하나만을 대상으로 멀티미디어 타이머와 기존 스케줄러에서 사용되는 sleep 방식을 사용한 경우 실제로 컴포넌트가 깨어나는 시간 주기를 측정하여 나타낸 것이다. 그림에서 보듯 기존의 스케줄링 방식보다 상대적으로 멀티미디어 타이머가 높은 정확성을 나타내지만, 멀티미디어 타이머는 약 9.76ms와 10.736ms에서 실제 인터럽트가 발생하는 것을 볼 수 있다. 앞서 3절에서 언급 했듯이 Windows NT의 멀티미디어 타이머는 최소 1ms의 해상도를 제공해 주지만 실제로는 976us에 인터럽트가 발생한다[7]. 우리는 실제 실험을 통해 10ms의 주기를 갖는 경우 실제로는 9.76ms의 인터벌을 가지고 인터럽트가 발생하고, 매 40ms에서는 10.736ms의 인터벌을 가지고 인터럽트가 발생하는 것을 확인하였다. 이 이유는 그림 6에서 보듯이 인터럽트 발생시간이 40번째는 40 * 0.976 = 39.04ms이고 41번째는 41 * 0.976 = 40.016ms가 되어 멀티미디어 타이머는 절대시간 40ms와 더 가까운 41번째 tick이 발생한 후에 인터럽트를 발생시키는 것을 알 수 있었다. 따라서 실험 결과 멀티미디어 타이머를 사용할 경우 10ms를 기준으로 -2.5% ~ + 7.5%의 오차는 피할 수 없음을 알 수 있다.

2. 외란(disturbance)에 의한 스케줄링의 주기성 검증

본 논문에서는 사용자 레벨의 스케줄러를 제안하고 있고,

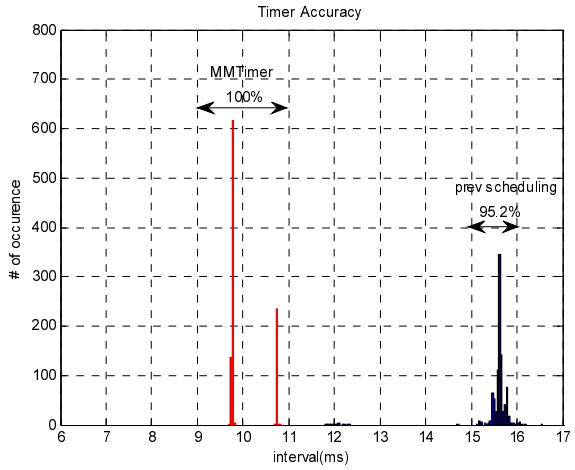


그림 6. 멀티미디어 타이머와 기존 스케줄링의 정확성.

Fig. 6. Timer accuracy of MMTimer and previous scheduling.

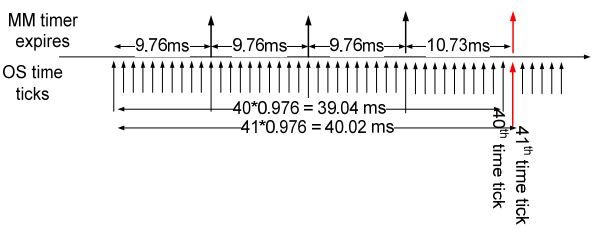
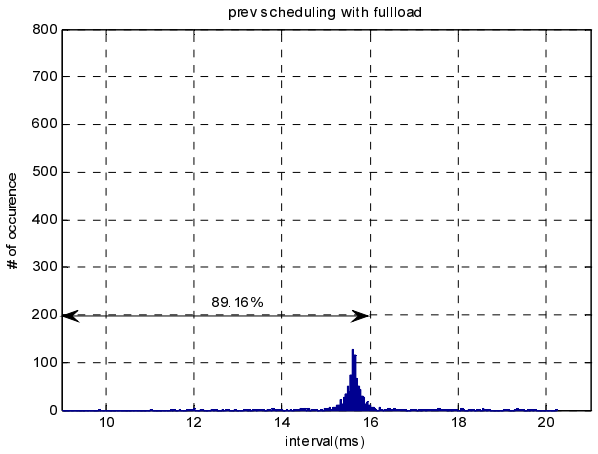


그림 7. 멀티미디어 타이머의 오차 원인.

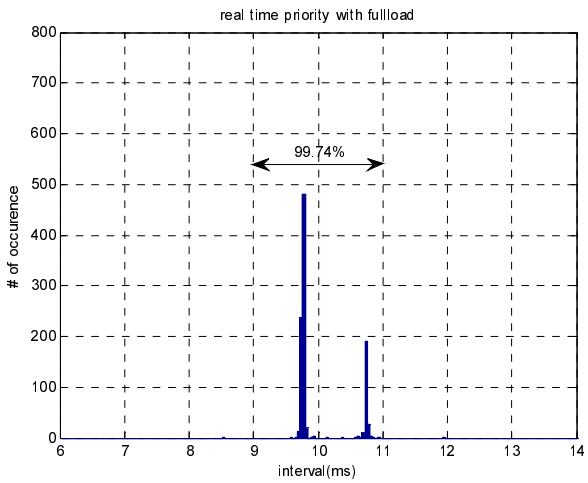
Fig. 7. Illustration of the periodic timer errors at the mm timer.

이 스케줄러는 커널 레벨의 프로그램 수행을 제어할 수는 없다. 하지만 범용 운영체제인 NT의 특성상 사용자의 의도와는 상관없는 커널 레벨의 프로그램이 수행될 가능성이 있다. 기본적으로 커널 레벨의 프로그램은 사용자 레벨 프로그램보다 높은 우선순위를 가지고 있기 때문에, 커널 레벨의 프로그램 수행에 의해 제안한 스케줄러의 지연이 발생 할 수 있다. 커널 레벨에 의한 스케줄링의 지연 정도를 검증하고자 ISR (Interrupt Service Routine), DPC (Deferred Procedure Call)등을 외란으로 정의하고 이에 따른 영향을 측정하는 실험을 수행하였다. 외란에 의한 영향은 분석적으로 계산할 수가 없기 때문에, 실험적으로 검증하였다. 그림 8은 일반 우선순위로 동작하는 기존의 스케줄러와 실시간 우선순위로 동작하는 제안된 스케줄러 환경에서 각각 외부 영향이 없는 경우와, 네트워크 데이터 수신(약 32Mbps), USB 트래픽 부하(약 100Mbps), 그리고 디스크 입출력의 외란을 발생시킨 경우 10ms 기준으로 스케줄러에 의해 실제 실행기가 스케줄링 되는 시간을 측정한 것이고, 표 1은 이 결과를 나타낸다.

실험결과 제안한 스케줄러 환경에서는 외란이 없는 경우 모든 스케줄링이 오차범위 10% 이내에 동작하는 것을 확인하였다. 또한 외란이 있는 경우에도 약 0.24% 정도의 오차



(a) 기존의 스케줄링



(b) 제안한 스케줄링

그림 8. 외란에 의한 스케줄링의 주기성.

Fig. 8. Scheduling interval variation of external disturbance.

표 1. 환경 변화에 따른 스케줄링 시간 정확성 분포.
Table 1. Scheduling accuracy with external disturbance.

제안한 스케줄러(REALTIME_PRIORITY_CLASS)			
	Max(ms)	Min(ms)	10%오차
No Load	10.82	9.05	100%
Full Load	13.1	6.42	99.74%
기존의 스케줄러(NORMAL_PRIORITY_CLASS)			
	Max(ms)	Min(ms)	60%오차
No Load	19.44	11.83	96.88%
Full Load	20.25	9.86	89.16%

발생률 정도만이 추가로 생성 되었다. 이에 반해 기존의 스케줄링의 경우 60% 오차 범위를 측정한 결과 외란이 없는 상황에서 약 3.12% 오차 발생률이 생성되었고, 외란시에는 약 7.72%의 오차 발생률이 추가로 생성되어, 오차의 범위 또한 기존의 스케줄링에 비하여 큰 차이를 보이는 것을 알 수 있었다. 이러한 차이를 보이는 이유는 일반 우선순위로 동작하는 기존의 스케줄러의 경우 DPC에 의한 지연 이외에도 운영체제에서 실행되고 있는 일반 우선순위의 다른 응용프로그램에 의한 지연이 추가되었기 때문으로 생각된다. 따라서 실시간 우선순위의 환경에서는 이러한 지연의 영향을 받지 않을 수 있고, DPC에 의한 영향 역시 0.24%정도로 크지 않음을 알 수 있었다.

3. DC 모터 제어 응용 실험

우리는 설계한 스케줄러의 성능을 검증하기 위해서 DC 모터의 PI 위치제어를 그림 9와 같은 환경하에 실험하였다. 이 실험에서는 10ms 주기를 갖는 위치제어 컴포넌트와 100ms 주기를 갖는 비전처리(vision) 컴포넌트를 대상으로 실험하였고, single core인 상황에서 실험하였다. 실험내용은 다음과 같다.

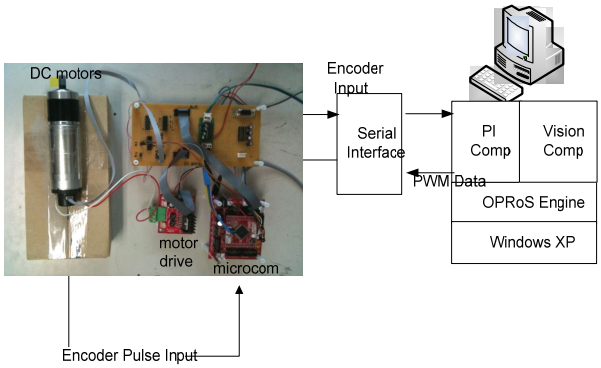


그림 9. DC모터 컨트롤의 셋업 환경.
Fig. 9. Experiment setup for DC motor control.

표 2. DC 모터 & PI 제어 파라미터.
Table 2. DC Motor & PI control parameter.

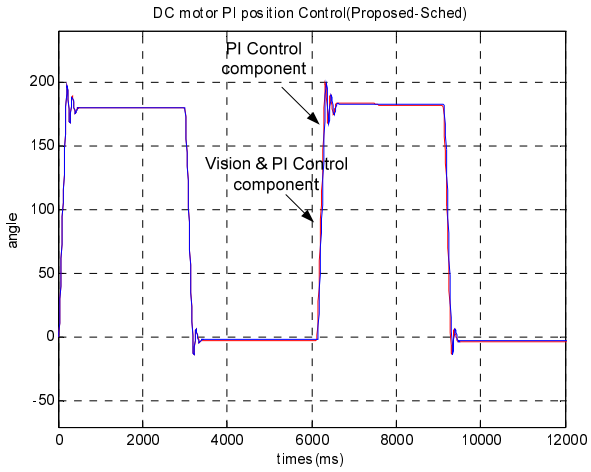
Parameter	Value
RPM	172rpm
엔코더 해상도	3584pulse/rotate
PI 제어 주기	100Hz(10msec)
PWM 제어 주기	20KHz(50usec)

우선 DC 모터의 엔코더 펄스가 AVR의 입력으로 들어간다. AVR은 10ms주기로 이 값을 UART를 통해 PC로 전송한다. PC 환경의 OPRoS Engine은 멀티미디어 타이머에 의해서 10ms마다 깨어나 위치제어 컴포넌트를 수행시키고, 이 컴포넌트는 3초마다 180도와 0도를 기준각도로 설정하고 UART로 데이터를 받은 뒤 PI 연산을 한 후 연산된 값을 다시 UART를 통해서 AVR로 전송한다. UART를 통해 새로운 데이터를 받은 AVR은 이 값으로 DC 모터의 위치제어를 수행한다. 그리고 100ms 주기마다 비전처리 컴포넌트가 영상 처리를 수행하게 된다. 이때 OPRoS Engine은 기존의 스케줄링 방식과 제안된 스케줄링 방식을 비교 실험 하였고, 앞서 언급한 외란은 모두 존재하는 상황에서 실험하였다.

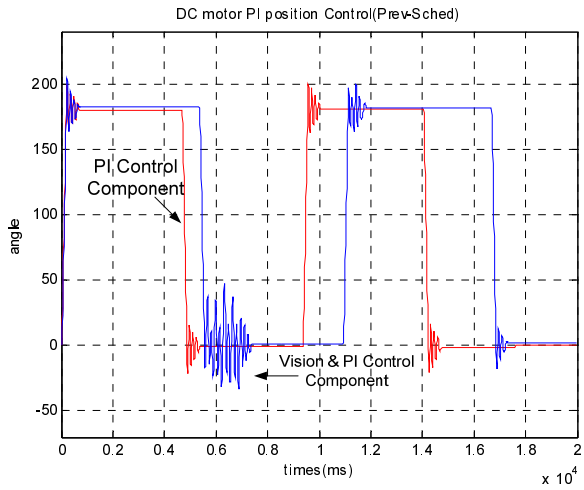
이때 사용된 PI 모터 제어 시스템의 파라미터는 표 2과 같다.

그림 10은 비전처리 컴포넌트의 유/무에 따른 DC 모터의 위치제어 결과 값을 나타낸다.

실험결과 제안된 스케줄링 방식의 경우 비전처리 컴포넌트의 유/무에 따른 영향이 거의 없고 비슷한 결과를 보이는 것을 확인하였다. 기존의 스케줄링 방식을 사용한 경우 비슷



(a) 제안한 스케줄링



(b) 기존의 스케줄링

그림 10. DC 모터 PI 위치 제어 결과 파형.
Fig. 10. DC motor PI control test.

한 결과를 내는 상황도 존재 하였지만, 오실레이션의 범위가 커지거나 오실레이션 기간이 더 길어지는 구간을 확인할 수 있었다. 이러한 이유는 비전처리 컴포넌트가 PI 컴포넌트보다 먼저 수행될 가능성이 있고, 이때 비전처리 컴포넌트의 실행시간이 PI 컴포넌트의 주기보다 길어지는 경우 PI 컴포넌트의 제어 주기를 잃기 때문으로 판단된다. 제안된 스케줄링의 경우 PI 컴포넌트가 비전처리 컴포넌트보다 항상 먼저 실행됨을 보장할 수 있고, 비전처리 컴포넌트가 실행 중에도 PI 컴포넌트가 선점 가능하기 때문에 주기성을 잃지 않는 것으로 파악된다. 이러한 결과로 미루어 제안된 스케줄러를 이용하여 위와 같은 제어 시스템에 적용이 가능하다고 판단된다.

VI. 결론

본 논문은 OPRoS 프레임워크가 Windows NT기반환경에서 실시간 스케줄링을 제공하기 위한 방안을 제안하고 이를 실험하였다. 실험결과 10ms의 경우 -2.5% ~ + 7.5%의 오차를 허용하는 경우에 충분히 실시간성을 제공할 수 있음을 확인하였다. 제안된 스케줄러는 Windows NT의 몇 개 안 되는 실시간 선점형 스레드를 활용하고, 운영체제의 시간 정밀도를 제어하며, 컴포넌트의 실시간적 특성을 고려하여 실시간성을 보장하도록 설계되었다.

반면 Windows NT 커널이 갖는 극복하기 쉽지 않은 한계점으로, 멀티미디어 타이머의 정확성의 제약, DPC 등의 외란 등이 존재하였다. 하지만 이러한 상황에서도 설계된 스케줄러는 외란의 영향을 약 0.2%정도로 줄일 수 있었으며 오차의 범위도 ±3%정도로 제한할 수 있다는 것을 알 수 있었다. 또한 외란이 있는 경우에서도 DC 모터의 PI 제어가 가능한 것을 확인 하였다. 따라서 실시간 운영체제로서의 부적한 점이 있음이 분명하지만, 서비스로봇이 필요로 하는 제어주기 정도와 응용 개발의 효율성 등을 고려할 때 Windows NT기반의 시스템의 활용가능성은 높다고 생각된다.

본 연구에서는 서비스로봇의 응용 예로 DC 모터를 사용한 바퀴의 위치제어의 결과를 보였다. 현재 역진자(inverted pendulum)와 실제 로봇에 적용하는 실험이 추가적으로 진행중이다. 또한 하나의 응용에 동일하지 않는 주기를 사용하는 컴포넌트에 대한 스케줄링방안에 대한 좀더 상세한 설계를 진행 중이다. 또한 본 논문에서는 HAL 영역에서 타이머의 부정확성을 인정하고 진행하였는데, 타이머의 정확성을 확보하기 위한 방안에 대해서도 알아보고 있는 중이다. 마지막으로 실시간성을 보장하려면 태스크들에 대한 수행허용 여부를 예측하고 이를 제어하여야 하는데, 현재는 모두 주기적인 실행을 갖는 것으로 가정하고, 고전적인 RMS 스케쥴링 이론[18]을 사용하였으나, OPRoS 시스템의 특성에 맞는 알고리즘을 개발해야 할 필요성도 있다. 또한, 실시간 리눅스[19]를 사용하는 경우와의 성능비교도 추후연구로 필요하며, 제안된 선점형기능을 이용하여 컴포넌트의 실행 오류 등에 적용[20,21]하는 방안에 대한 연구에 접목을 구체화 과정에 있다.

참고문헌

[1] KS B 6939, “Service robots-Vocabulary-Part1: Classification and general definition” 2006.

[2] S. Thrun, M. Bennewitz, W. Burgard, A. B. Cremers, F. Dellaert, D. Fox, D. Hahnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz, “MINERVA: A second-generation museum tour-guide robot,” *Proc. of the Int. Conf. on Robotics and Automation (ICRA ’99),* 1999.

[3] Y. Sakagami, R. Watanabe, and C. Aoyama, “The intlelligent ASIMO: system overview and integration,” *Proc. of ICIRS, Swiss,* 2002.

[4] Korean Intelligent Robot Standard Forum, “OPRoS Component Spec.” Draft, 2009.

[5] OMG, Robotic Technology Component Specification Version 1.0, April 2008.

[6] Microsoft Robotics Developer Studio, [online] Available: http://www.microsoft.com/Robotics/

[7] M. Timmerman, “Windows NT as Real-Time OS,” *Real-time Magazine,* 1997.

[8] M. B. Jones, “The problems you’re having may not be the problems you think you’re having: results from a latency study of windows NT,” *In Proc. of the 7thWorkshop on Hot Topics in Operating Systems,* Rio Rico, AZ, pp. 96-101, 1999.

[9] A. Baril, “Using Windows NT in Real-Time Systems,” *Proc. of Fifth IEEE Real-Time Technology and Applications Symposium (RTAS'99),* Vancouver, Canada, pp. 132-137. 1999.

[10] O. Gonzáles, S. Sen, S. Shirgurkar, C. Shen, and K. Ramamrithham, “Using windows NT for real-time applications: experimental observations and recommendations,” *Proc. of Fourth IEEE Real-Time Technology and Applications Symposium (RTAS'98),* Vancouver, Canada, pp. 102-107, 1998.

[11] C. Lee and C. Mavroidis, “WinReC v.1: Real-Time Control Software for Windows NT and its Applications,” *Proc. of the 2000 American Control Conference,* Chicago, IL, pp. 651-655. 2000.

[12] B. Carpenter, M. Roman, N. Valsilatos, and M. Zimmerman, “The RTX real-time subsystem for windows NT,” *Proc. of USENIX Windows NT Workshop,* Seattle, Washington, Aug. 1997.

[13] OPRoS project official site, http://www.opros.or.kr/

[14] A. Gambier, “Real time control system: A tutorial,” *Proc. of 5th Asian Control Conference,* 2004, vol. 2, pp. 1024-1031, 2004.

[15] H. Custer, Inside Windows NT, Microsoft Press, 1992.

[16] J. J. Labrosse, *MicroCOS II: The Real time Kernel,* CMP Books, 2002.

[17] Jane W. S. Liu, *Real-Time Systems*, Prentice Hall PTR, 2000.

[18] J. Lehoczky, L. Sha, and Y. Ding, “The rate monotonic scheduling algorithm: Exact characterization and average case behavior,” *IEEE Real-Time Systems Symposium,* pp. 166-171, 1989.

[19] E.-C. Shin and B.-W. Choi, “Implementation of a mobile robot control platform using real-time embedded linux,” *Journal of Institute of control, Robotics, and Systems*, vol. 12, no. 2, pp. 194-200, 2006.

[20] H. Ahn, D.-S. Lee, and S.-C. Ahn, “OPRoS based fault tolerance support for reliability of service robots,” *Journal of Institute of control, Robotics, and Systems*, vol. 16, no. 6, pp. 601-607, 2010.

[21] H. Ahn and S.-C. Ahn, “State-monitoring component-based fault-tolerance techniques for OPRoS framework,” *Journal of Institute of control, Robotics, and Systems,* vol. 16, no. 8, pp. 780-785, 2010.

부록

부록에는 본 논문에서 제안한 스케줄러를 코드의 형태로 표현한다. 코드는 Microsoft Windows C style로 표현하였고, 지면 관계상 중요 부분을 위주로 발췌하였다.

1. 멀티미디어 타이머와 OS tick setting

본 절에서는 'MMTimerSetup()'과 'MMTimerDestroy()' 이 두 두 함수를 통하여 멀티미디어 타이머와 OS tick 설정 및 해제를 나타낸다.

```
void MMTimerSetup() {
    //get the smallest OS time resolution
    TIMECAPS tc;
    timeGetDevCaps(&tc, sizeof(TIMECAPS));
    wTimerRes =
    min(TARGET_RESOLUTION), tc.wPeriodMax);
    timeBeginPeriod(wTimerRes);//set OS tick

    wTimerID =
    timeSetEvent(
        msInterval, //desired interval(10ms)
        wTimerRes,
        hEventCPU[NUMCPU],//timer event
        (DWORD_PTR)npSeq,//user data
          TIME_PERIODIC |
          TIME_CALLBACK_EVENT_SET |
          TIME_KILL_SYNCHRONOUS );
}

int MMTimerDestroy()
{
    // destroy MMTimer
    timeKillEvent(wTimerID);

    //restore OS tick
    timeEndPeriod(wTimerRes);
    return 0;
}
```

2. 스케줄링 스레드

스케줄링 스레드는 멀티미디어 타이머에 의한 이벤트나 실행기가 호출하는 'yield_cpu()'에 의한 이벤트 모두를 기다린다. 멀티미디어 타이머에 의한 이벤트인 경우엔 모든 cpu core에 대한 스케줄링을 수행하고, yield_cpu()에 의한 이벤트인 경우 yield_cpu()를 호출한 실행기가 속한 cpu core에 대해서만 스케줄링을 수행한다.

```
DWORD WINAPI SchedulerFunc(LPVOID arg)
{
  //Set RealTimePriority Process
    HANDLE hProcess = GetCurrentProcess();
    SetPriorityClass
    (hProcess, REALTIME_PRIORITY_CLASS);

    //make scheduler high priority
  SetThreadPriority(GetCurrentThread(),
      THREAD_PRIORITY_TIME_CRITICAL);
    int nObjects = NUMCPU + 1;//# of CPU + 1

    while(1) {
      DWORD eType =
      WaitForMultipleObjects(nObjects, hEventCPU,
      FALSE, TIME_FOR_DEADLOCK);
      if(eType < WAIT_OBJECT_0 + NUMCPU) {
        // executor call 'yield_cpu()'
        schedule_user_cpu(eType - WAIT_OBJECT_0);
      }else if(eType == WAIT_OBJECT_0 + NUMCPU) {
        //mm timer interrupt
        timer_interrupt();
      }
    }
    return 0;
}

void timer_interrupt(void)
{
    int i;
    EnterCriticalSection(&cs_schedule);
    update_timers();

    for(i=0; i < NUMCPU; i++) {
      schedule_cpu(i);
    }
    LeaveCriticalSection(&cs_schedule);
}

bool schedule_cpu(int core)
{
    TCB_t *pToRun = getHighestReady(core);
    if(pToRun == NULL) {
      //no one want to run or cur is highest prio
      return true;
    }

    if(RunningExecutor[core] == NULL) {
      //no one is running yet.
      RunningExecutor[core] = pToRun;
      pToRun->status = st_running;
      ResumeThread(pToRun->handle);
    }

    else if(pToRun->period <
    RunningExecutor[core]->period) {
      //new one is more high priority
      SuspendThread(RunningExecutor[core]->handle);
      RunningExecutor[core]->status = st_wait;

      RunningExecutor[core] = pToRun;
      pToRun->status = st_running;
      ResumeThread(pToRun->handle);
    }
    return true;
}

void yield_cpu(TCB_t *pTCB) {
    //executor want to yield CPU

    EnterCriticalSection(&cs_schedule);
    pTCB->status = st_wait;//setting waiting for sync
    LeaveCriticalSection(&cs_schedule);

    SetEvent(hEventCPU[pTCB->cpu]);
```

```
        //wakeup scheduler for yield CPU

         //busy while until scheduler make me suspend
        while(1) {
           if(pTCB->status == st_running)
                break;
           Sleep(0);
        }
}
```

3. 실행기

본 절에서는 제안한 스케줄러를 사용하기 위한 실행기의 알고리즘을 간단히 나타내었다. 실행기는 컴포넌트들을 동작시킨 후 스스로 sleep상태에 빠지지 않고, yield_cpu()를 호출하여 스케줄러에 의하여 suspend상태로 변경된다.

```
void Executor::run(){
        while(m_runningState == OPROS_ES_ACTIVE){
           executeComponents();
           Scheduler()->yield_cpu(p_TCB);
        }
  }
```

이 동 수

2009년 서울산업대학교 제어계측공학과(공학사). 2009년~현재 서울과학기술대학교(구 서울산업대학교)산업대학원 제어계측공학과 재학중. 관심분야는 임베디드 소프트웨어, 실시간 시스템, 로봇공학.

안 희 준

1993년 KAIST 전기및전자공학과(공학사). 1995년 KAIST 전기및전자공학과(공학석사). 2000년 KAIST 전기및전자공학과(공학박사). 1999년~2000년 독일 뉴른버그대학 박사후연구원. 2000년~2003년 LG 전자-정보통신 선임연구원. 2002년~2003년 Tmax Soft 연구소 책임연구원. 2004년~현재 서울과학기술대학교 제어계측공학과(부교수). 관심분야는 소프트웨어 최적화, 실시간 시스템, 영상 통신, 인터넷 프로토콜.